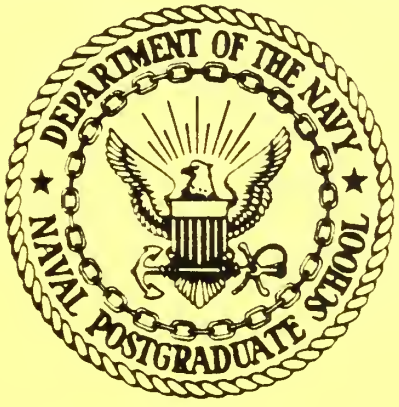
NPS52-82-001

NAVAL POSTGRADUATE SCHOOL

Monterey, California



ON ROBUSTNESS OF DEADLOCK DETECTION ALGORITHMS FOR DISTRIBUTED COMPUTING SYSTEMS

Dushan Z. Badal and Michael T. Gehl

February 1982

Prepared for:

Naval Postgraduate School
Monterey, California 93940

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

David A. Schrady
Acting Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

GORDON H. BRADLEY, Chairman
Department of Computer Science

WILLIAM M. TOLLES
Dean of Research

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: NPS52-82-001	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): On Robustness of Deadlock Detection Algorithms for Distributed Computing Systems	5. TYPE OF REPORT & PERIOD COVERED: Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Dushan Z. Badal and LCDR Michael T. Gehl, USN	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61152N; RR000-01--10, N0001482WR20043
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93940	12. REPORT DATE: February 1982 13. NUMBER OF PAGES: 35
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This paper has been submitted for publication and if accepted it will be copyrighted for publication. It has been issued as a technical report for early dissemination of its contents.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Robustness, Reliability, Deadlock Detection, Distributed Computing Systems, Failures.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In this paper we investigate the robustness of several deadlock detection algorithms for distributed computing systems. We analyze the behavior of each algorithm in the presence of two classes of failures - lost messages and single site failures. In the case of single site failure we consider six different types of sites depending on how they can participate in deadlock and deadlock detection. The observation and conclusions made in this

paper are intended to show how robust the present algorithms are and to provide an insight and better understanding of distributed algorithms robustness.

ON ROBUSTNESS OF DEADLOCK DETECTION ALGORITHMS FOR DISTRIBUTED COMPUTING SYSTEMS

Dushan Z. Badal and Michael T. Gehl
Computer Science Department
Naval Postgraduate School
Monterey, California 93940

Abstract

In this paper we investigate the robustness of several deadlock detection algorithms for distributed computing systems. We analyze the behavior of each algorithm in the presence of two classes of failures - lost messages and single site failures. In the case of single site failure we consider six different types of sites depending on how they can participate in deadlock and deadlock detection. The observation and conclusions made in this paper are intended to show how robust the present algorithms are and to provide an insight and better understanding of distributed algorithms robustness.

I. INTRODUCTION.

There have been many algorithms published for deadlock detection, prevention or avoidance in centralized multiprogramming systems. The problem of deadlock in those systems has been essentially solved. In the past decade there has been considerable work done on distributed computer networks and multiprocessor systems. Both of these are predecessors of distributed computing systems which are presently a focus of intensive research and development in academia and industry. Many techniques for concurrency control, reliability/recovery or security developed for centralized (or single CPU) systems have been or are being adopted and adapted for distributed computing systems. For example, there is a tendency to use locking as a general synchronization technique in distributed systems and its special variant, two-phase locking, for distributed database systems. Up until recently it has been argued that the frequency of deadlock occurence in existing applications is so low that the problem of deadlock in distributed systems is not very important and therefore can be managed by adopting techniques developed for centralized systems. However, it has become recently apparent that deadlocks may be a problem in the future as we see new applications featuring large processes and/or many concurrent processes or transactions[GRA81]. As an example of such new applications we mention information utility systems which service concurrently hundreds or perhaps thousands of TV users.

The distributed computing systems are characterized by the absence of global memory and by message transmission delays which are not negligible. Additionally, the processes operating at the same or different sites can communicate with each other, and can share resources. If locking is used as the synchronization technique, then

the last two items raise the problems of deadlock occurence in distributed systems, and the first two characteristics of distributed systems make it much more difficult to detect, avoid or prevent than in the earlier multiprogramming centralized computing systems.

Deadlock prevention and avoidance algorithms for a distributed computing systems are not efficient. Prevention can be accomplished by not allowing concurrent processing, by assigning priorities and allowing preemption, by requiring a process to acquire all resources it will need before it starts, or by having no locks. Requiring sequential execution in a distributed system is a gross waste of resources. Having prioritized processes will result in lower-prioritied processes being restarted many times, with a major degradation in system efficiency. Dynamic prioritization would be a complex algorithm by itself. A process may be unable to determine its minimum set of resources, and therefore would have to acquire the set of all probable and possible resources, even though it may not need them. In addition, in systems in which messages are treated as resources, it is impossible to determine in advance which messages will be required. Having no locks may result in database inconsistencies, assuming a non-optimistic concurrency controller. Similarly, deadlock avoidance algorithms, which either calculate a 'safe path' [GOL77] or never wait for a lock[GRA78] are also inefficient. Safe path algorithms require a non-trivial execution time, and must be done each time a resource request is to be granted. Never waiting for a lock is inefficient when deadlock is a rare occurence. Thus, in distributed computing systems, deadlock detection and resolution algorithms must be used.

There are four criteria that any deadlock detection algorithm for distributed computing systems must meet. They are 1) correctness, 2)

robustness, 3) performance, and 4) practicality. Correctness refers to the ability of the algorithm to detect all deadlocks, and the ability to not detect any false deadlocks. Robustness refers to the ability of the algorithm to be correct even in the presence of anticipated faults. This includes the ability to detect deadlocks even when a site fails or loses communications while the deadlock detection algorithm is being executed. The performance of the algorithm refers to its overhead - the delays between deadlock and detection, CPU time used, number of messages required, etc. Practicality is closely related to performance. It refers to aspects such as complexity and cost.

Several different approaches are being used in current deadlock detection and resolution algorithms for distributed systems. Two major ones are centralized and distributed deadlock detection algorithms. Within the distributed class are two subclasses; 1) all or several sites execute the deadlock detection algorithm, and 2) only one site is actually executing, although the algorithm is resident in all sites and thus any site could execute the algorithm. It might be easier to view the algorithms as a continuum: fully centralized[GRA78], hierarchical[MEN79], distributed with a single site at a time executing the algorithm[GOL77], distributed with all sites involved in a possible deadlock executing the algorithm concurrently[MEN79], and distributed with all sites executing the algorithm concurrently[ISL78].

In this paper we investigate the robustness of several published deadlock detection and resolution algorithms for distributed systems. The motivation for our work comes from three facts. First, very few authors investigated robustness or reliability of deadlock detection

algorithms. Second, reliable deadlock detection and resolution for upcoming new distributed systems and applications is in our opinion an urgent, very important and as yet not satisfactorily resolved problem. Third, as there can be more than one deadlock being detected by the deadlock detection algorithm then it is reasonable to expect such algorithm to be robust, i.e., to continue executing and detecting all deadlocks even in the presence of failure(s) which might have in effect breaked one of the deadlocks being detected.

The paper is organized as follows. In section two, we discuss robustness of distributed systems. In section three, we analyze the robustness of several existing deadlock detection algorithms with respect to some single failures. In section four, we present our conclusions based on the analysis of section 3.

II. SOME THOUGHTS ON ROBUSTNESS IN DISTRIBUTED SYSTEMS.

In this paper we want to investigate the robustness of deadlock detection algorithms (DDA), i.e., we want to find out the impact of some single failures on such algorithms. In general, the DDA is invoked by two events - either whenever a process waits for a resource, or after a certain period of time has elapsed since the last DDA invocation. In the first case, deadlock is checked for whenever its possibility appears, and in the second case it is checked for periodically (i.e., regardless of whether its possibility exists).

The DDA can reside in one, several or all sites of the distributed computing system. When a triggering event for DDA occurs, then depending on a particular algorithm one, several or all sites will receive information from several or all sites. Such information consists of "who waits for whom and where", and it can be represented by

arcs of the wait-for graph, strings, or lists of processes or transactions. Upon receipt of such information one, several or all sites attempt to reconstruct a global state of the distributed system, i.e., to generate a true snapshot of all or of all waiting processes in the system.

The generation of such a true snapshot in the distributed system is difficult because of lack of global memory and the message delays which are not neglibible and can vary considerably. The generation of such a true snapshot, usually referred to as a global wait-for graph, becomes even more difficult when we consider a possibility of failures in the distributed system. Some system mechanisms have been designed to be robust or reliable. For example, some concurrency control or synchronization mechanisms for distributed databases and transaction processing systems are based on two phase locking, which has been made robust by incorporating atomicity by using two phase commit protocols. The two phase commit protocol supports not only the atomicity of transactions but also it supports the robustness of locking, i.e., the robustness of concurrency control mechanisms. In particular what makes the concurrency control which uses locking robust is the need to lock and unlock resources in a robust way, i.e., either all lock/unlock operations for a given process or transaction occur or none occur. Thus in some sense, the robustness of concurrency control is meant to support the atomicity of placing and releasing a set of locks needed by a process. In other words, the robustness of concurrency control means that no dangling locks or locked resources are left behind the terminated or committed process, even in the presence of some failures. It is interesting to note that although deadlock detection is a part of concurrency control based on locking, there has

been no attempt to provide for or even to investigate the robustness of deadlock detection mechanisms. The most likely explanation for this is that from the concurrency control point of view, the inability of the process to lock a needed resource is an exception to be handled by another mechanism, i.e., a deadlock detection algorithm (DDA).

The proper way to see the DDA is as another transaction running under the concurrency control mechanism, as it reads and shares lock tables with concurrency controllers and other transactions. However, DDA is a special transaction which operates on special data it creates solely for deadlock detection, e.g., wait-for graphs. Such data, we'll call it deadlock data, is internal to each invocation of DDA transaction and is erased after its execution. Moreover, such deadlock data is not shared by any other DDA transaction invocations and therefore they need not be locked. This means that the robustness required of DDA transactions is of a somewhat different kind than the robustness of transactions operating on shared database data. Thus it makes sense that the DDA transaction does not need to use two phase commit to assure its robustness. The question then is what kind of robustness or fault-tolerance we need for DDA transactions and this is precisely the problem we are addressing in this paper.

We consider the following informal model of DDA transaction execution. The DDA is invoked by a concurrency controller at a site at which a database transaction can not acquire locks which are being held by another transaction(s). The DDA transaction executes at one, several or all sites (depending on the DDA itself and the deadlock topology). During its execution the DDA transaction should exhibit the atomicity property, i.e., it either executes correctly or it does not execute at all. The results of DDA transaction execution are two

messages to the concurrency controller which has triggered it:

1) Proceed - because of
   a) no deadlock
   b) deadlock detected but another transaction was selected as a victim for back-up

2) Abort - because of
   a) deadlock detected and you are the victim.
   b) DDA transaction failed, i.e., it did not execute.

The situation we investigate in this paper is when DDA transactions fail or should not fail, i.e., how robust the existing DDA's are or should be. In this paper we consider only two classes of single failures. First, we investigate the impact of lost messages and second, we investigate the impact of one site failures, or identically one site partitions on DDA behavior. We investigate the impact of lost messages because not all distributed systems may support reliable delivery of messages, several algorithms treat messages as resources[GOL77], and in some applications, acknowledgements cannot be sent.

III. RELIABILITY ANALYSIS OF DEADLOCK DETECTION ALGORITHMS

In this section, we examine four published deadlock detection algorithms for distributed computing systems with respect to the presence of the two classes of failures (lost messages and site failures) discussed in section two. Although very few of them have already been shown to be correct when no failures or errors occur, we feel that their robustness is nevertheless worth analyzing. The assumptions made by each author will be discussed in the context of how robust the algorithm is. We will analyze each DDA by executing it in the following environment.

There are four sites in the system, each of which has a single

resource and a single transaction. (These restrictions merely make the example simpler, they are not required for the analysis.) The initial system status is shown in figure 1. Transaction T1 at site A holds resources R2 and R3 and is waiting for resource R4. Transactions T2 and T3 hold no resources. Transaction T4 at site D holds resource R4, but is active. We assume that the deadlock detection activity resulting from T1 waiting for R4 has been completed, so there is currently no deadlock detection activity in the system. For the algorithms which require global timestamps, we assign timestamp (TS) t1 to the T1<—R2 assignment, t2 to the T4<—R4 assignment, t3 to the T1<—R3 assignment, and t4 to the T1—>R4 request. Now at some time t6, transaction T4 requests R3, resulting in a global deadlock T1—>T4—>T1.

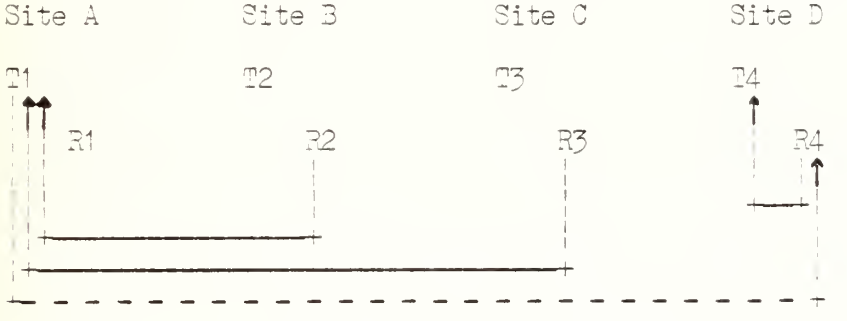


Figure 1

In the case of a site failure, we distinguish the following cases. a) A site can have a transaction involved in a deadlock but not be involved in deadlock detection, b) a site can have a transaction involved in a deadlock and be involved in detection, c) a site can have a resource involved in a deadlock and not be involved in detection, d) a site can have a resource involved in a deadlock and be involved in a detection, or e) a site can be involved in deadlock detection but in no way involved in a deadlock. Not all of these possibilities exist with each algorithm.

A. THE DISTRIBUTED DEADLOCK DETECTION ALGORITHM OF GOLDMAN.

In [GOL77], Goldman presents two deadlock detection algorithms. Only the distributed version will be considered in this paper. A Process Management Module (PMM) at each site handles resource allocation and deadlock detection. An 'ordered blocked process list' (OBPL) is a list of process names, each of which is waiting for access to a resource assigned to the preceeding process in the list. The last process in the list is either waiting for access to the resource named, or it has access to that resource. An OBPL is created each time a PMM wants to see if a blocked process is involved in a deadlock. In the distributed algorithm, an OBPL is passed from a PMM to another PMM which has information either about a resource or a transaction in the OBPL which is needed to expand the OBPL. Each PMM adds the information it knows, and either detects a deadlock, detects a non-deadlocked state, or passes the OBPL to another PMM for further expansion. The terms process and transaction will be used synonymously in the analysis of this DDA. If several transactions are waiting on one transaction, multiple copies may be made of the OBPL and sent to each site having one of those waiting transactions. Processes can be in either of 2 states, active or blocked (waiting). A blocked process could be waiting for a database object, message text from another process or message text from an operator. A process is active if it is not blocked. In the algorithm, PX and RX are temporary variables representing a process or resource. The steps of the algorithm are:

1. Set RX to the value contained in the resource identification portion of the OBPL. If RX represents a local resource, go to 2. Otherwise, go to 3.

2. Verify that the last process added to OBPL is still waiting for RX. If so, go to 3, otherwise, halt.

3. Let PX be process controlling RX. If PX is already in OBPL, then there is a deadlock. If not, go to 4.

4. If PX is local to current PMM, go to 5, otherwise go to 7.

5. If PX is active, there is no deadlock. Discard OBPL and halt. Otherwise go to 6.

6. Add PX to OBPL and go to 10.

7. Add PX and RX to OBPL. Send OBPL to PMM in site in which PX resides. Halt.

8. Verify that last process in OBPL still has access to RX. If not, there is no deadlock, so discard OBPL and halt. If so, go to 9.

9. If last process in OBPL is active, there is no deadlock, so discard OBPL and halt. Otherwise go to 10.

10. Call resource for which last process is waiting RX. If RX is local, go to 3. Otherwise go to 11.

11. Place RX in OBPL and send OBPL to PMM of site in which RX resides. Halt.

Figure 2 shows the actions taken at each site during the execution of the DDA following the request by T4 for resource R3. The numbers refer to the current step being executed by the DDA. As can be seen, the algorithm correctly detected the resulting deadlock, in an environment of no faults. If, however, a message is lost (in our example, either the OBPL sent from site C to A, or the OBPL sent from A to D), the necessary information to detect the deadlock will be lost, and the algorithm will fail to detect an existing deadlock.

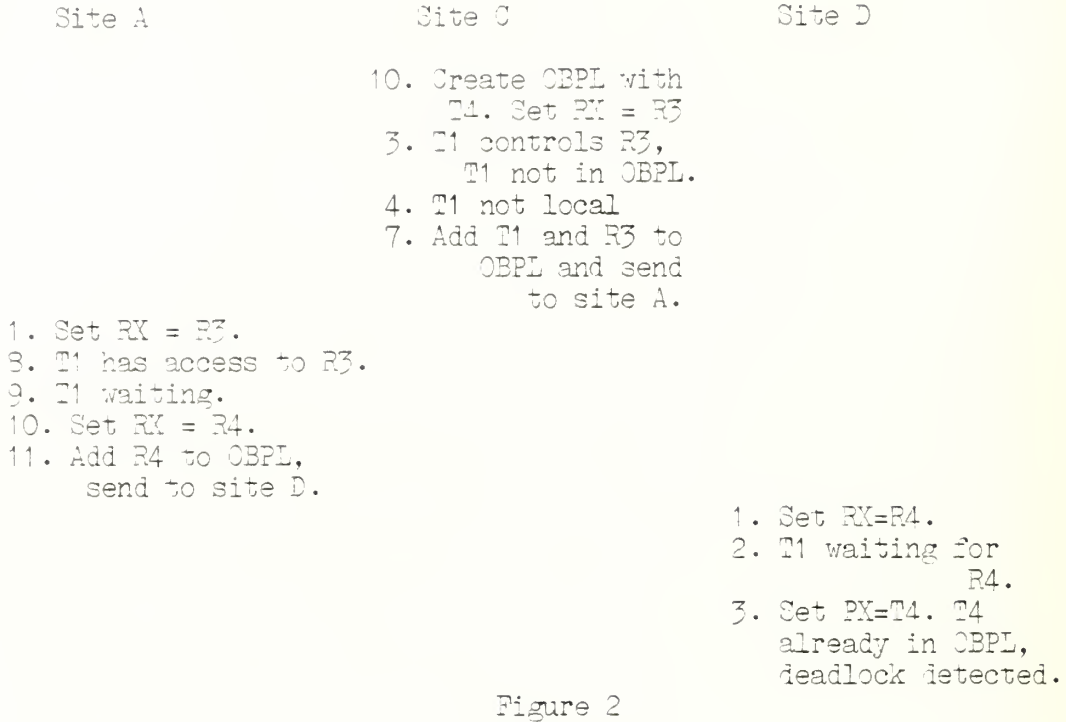


Figure 2

Goldman's algorithm allows the following types of sites discussed previously: type b (a site can have a transaction involved in deadlock and the site is involved in detection), type d (a site can have a resource held by a transaction involved in deadlock and the site will be involved in deadlock detection), and type c (a site can have a resource held by a transaction involved in a deadlock and not be involved in deadlock detection). A site could also be in several of the categories above, depending on the complexity of the system state. For example, site D could be considered a type b or type d site. If a site of type b (sites A or D in our example) fails during execution of the DDA, the behavior could be different depending on the time of the failure. If the failure occured at site A before site C sent the OBPL to site A, site C would realize that site A had failed. The algorithm includes no procedure for this occurence, so the behavior would be dependent on the underlying system. If the failure at site A occured after it received the OBPL, all deadlock detection

activity will cease, because only site A was currently involved in deadlock detection. A system timeout mechanism would eventually abort the transactions involved in the deadlock. A failure at site D would have the same effect as at site A.

If a site of type d (site C in our example) failed, the time of the failure would again determine the behavior of the DDA. If the failure occured before site C sent the OBPL to site A, deadlock detection activity would cease without deadlock having been detected. If the OBPL had been sent, however, deadlock detection would continue at sites A and D (sequentially) with site D detecting a deadlock. The failure of site C would not have been critical after the OBPL had been sent. The effect of a type c site (site B in our example) failing would have no effect on the behavior of the DDA, because the fact that R2 is held by T1 is not used or known by the DDA at any site.

There are essentially two types of OBPL's created by this DDA. The first type, call it W, is when a process is waiting, but is not involved in a deadlock. This OBPL is subsequently discarded. The second type, call it D, is one which will eventually show a deadlock cycle. If there are n transactions involved in a deadlock cycle, this DDA will create from 1 to n type D OBPL's. In our example, only one was created. If the request by T1 for resource R4 hapened simultaneously with the request by T4 for resource R3, two OBPL's would have been created which would have resulted in two sites independently detecting the same deadlock, vice the one site in our example. Thus the robustness of this algorithm with respect to a single site failure is related to the ratio of the number of D type OBPL's created to the number of transactions involved in the deadlock. This ratio is however determined by the sequencing or timing of transactions messages

blocked resources. Such sequencing is of random nature. A ratio of 1 would provide the highest degree of robustness. When only a single OBPL is created, the robustness of the DDA is very similar to that of a centralized DDA; a single site failure can stop deadlock detection activity. We conclude that the robustness of this DDA can be analyzed but it can not be predicted.

B. THE MENASCE-MUNTZ DISTRIBUTED ALGORITHM

In [MEM79], Menasce and Muntz presented a distributed deadlock detection algorithm. Gligor and Shattuck [GLI80] presented a counter example which showed the algorithm to be incorrect in that it failed in some cases to detect a deadlock. They also proposed a modification to the algorithm which they thought would make it correct, but they felt the algorithm was impractical. In [TSA82], Tsai and Belford show that the algorithm as modified by Gligor and Shattuck is also incorrect. Nevertheless, we will investigate the enhanced algorithm (i.e., its modified version as suggested by Gligor and Shattuck) in the presence of errors.

The algorithm constructs a Transaction-Waits-For (TWF) graph at originating sites of transactions which are potentially involved in the deadlock being detected, and at sites at which some transaction could not acquire a resource. Nodes in the WF graphs represent transactions. An edge (Ti,Tj) indicates that transaction Ti is waiting for transaction Tj. A non-blocked transaction is a transaction that is not waiting and is represented in the TWF graph by a node with no outgoing arcs. A blocked transaction is waiting for some transaction to finish. A 'Blocking set' is defined as the set of all non-blocked transactions which can be reached by following a directed path in the TWF graph starting at the node associated with transaction T[MEM79].

A pair (T,T') is a 'blocking pair' of T if T' is in the blocking set of T. A 'Potential Blocking set' consists of all waiting transactions that can be reached from T [GLI80]. Sorig(T) means the site of origin of transaction T. Sk is the site currently executing the algorithm. The rules which define the enhanced algorithm, as executed as site Sk, are:

Rule 0: When a transaction T requests a nonlocal resource it is marked 'waiting'.

Rule 1: The resource R at site Sk cannot be allocated to transaction T because it is held by T1, ...,Tk. Add an arc from T to each of the transactions T1,...,Tk. If there is then a cycle formed in the TWF graph, deadlock has been detected. Otherwise, for each transaction T' in blocking set(T), send the blocking pair (T,T') to Sorig(T) if Sorig(T) =/= Sk and to Sorig(T') if Sorig(T') =/= Sk. Form a list of potential blocking pairs associated with T.

Rule 2: A blocking pair (T,T') is received. Add an arc from T to T' in the TWF graph. If a cycle is formed, then a deadlock exists.

Rule 2.1: If T' is blocked and Sorig(T) =/= Sk, then for each transaction T" in the blocking set(T'), send the blocking pair (T,T") to Sorig(T") if Sorig(T") =/= Sk.

Rule 2.2: If T is waiting and Sorig(T) = Sk, then for each potential blocking pair (T",T) send the blocking pair (T",T) to Sorig(T") if Sorig(T") =/= Sk. Then, discard the potential blocking pairs (T",T) and erase the 'waiting' mark of T.

Figure 3 shows the actions taken at each site during the execution of the DDA following the request by T4 for resource R3. As can be seen, the deadlock was correctly detected by site A, in absence of failures. If the request message (T4,R3) from site D to site C was lost, however, deadlock detection activity would cease. If the blocking pair (T4,T1) from site C to site D was lost, site A would still detect the deadlock. If, however, the blocking pair (T4,T1) from site C to site A was lost, site D would apply rule 2. Neither rule 2.1 or 2.2 applies, so deadlock detection activity would cease.

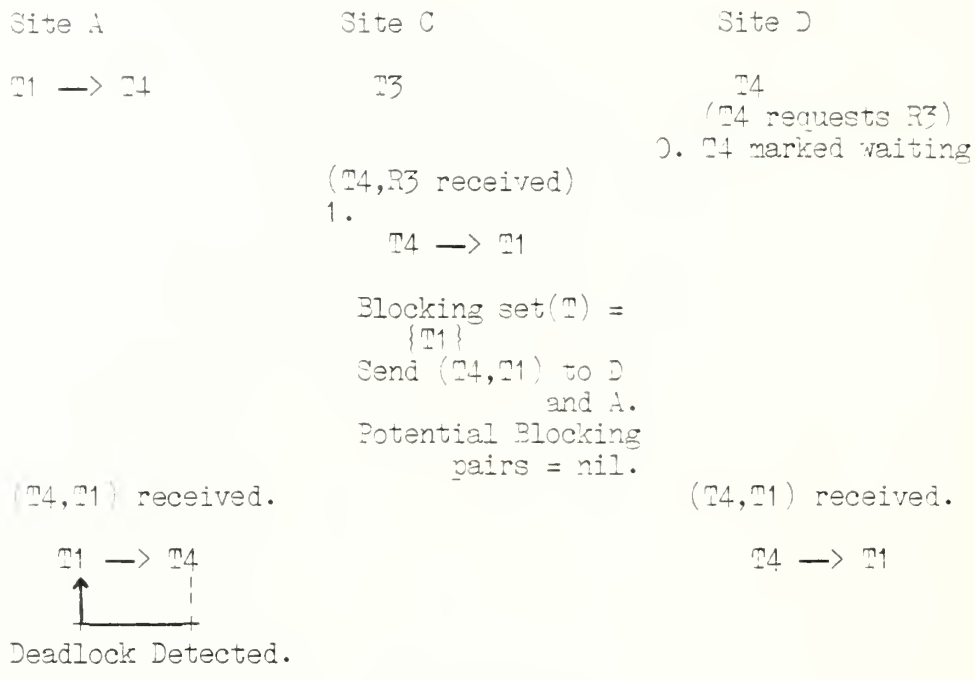


Figure 3.

This algorithm allows sites of types b, c, d and e, although our example does not include a site of type e. If a type b site (one having a transaction involved in the deadlock and the site is also involved in detection) failed, in our example site A (or site D), the behavior of the algorithm is dependent on the time of failure. If site A failed before receiving the blocking pair (T4,T1), site C would recognize the failure, but its action is not specified in the rules of the DDA. Site D would not detect the deadlock for the same reson as if the message from site C to site A was lost. If, however, the failure occured after site A received the blocking pair, deadlock detection activity would continue (at site D) but deadlock would not be detected. A failure of site D, also a type b site, at any time, would have no effect on detecting the deadlock in this example. If a type c site failed (site B), it would have no effect on detecting the deadlock. If a type d site (site C) failed, the time of its failure would determine the behavior of the DDA. If it failed before sending

the blocking pair to sites A and D, deadlock detection activity would cease. If it failed after sending those messages, it would have no effect on detecting the deadlock.

For our example, this algorithm behaved surprisingly similarly to Goldman's algorithm in almost all types and timings of failures. This may just be an anomaly found in small deadlock cycles, because in longer and more complex scenarios, it would appear that more sites would be involved in detection, and that there would be some duplication of information. As the number of transactions and resources) involved in a deadlock cycle increases, more blocking pairs and potential blocking pairs will be sent to more sites, i.e., the number of sites detecting the deadlock is increasing with the number of transactions involved in the deadlock and with the deadlock topology (or complexity). Thus there will be more chance of a deadlock being detected, as more parallel detection activity will be in progress. It appears, then, that as the site and complexity of deadlock increases, the robustness of this algorithm increases. However, as pointed out by Gligor and Shattuck, the effect which Gligor and Shattuck point out of rule 2.2 discarding information too early may have some impact on the increased robustness.

C. OBERMARCK'S DISTRIBUTED DEADLOCK DETECTION ALGORITHM.

In [OBE80], Obermarck presents a distributed deadlock detection algorithm. A centralized algorithm is presented by Obermarck and Beeri in [BEE81], but it is not discussed here because no mention is made in that paper about a backup capability if the site containing the centralized deadlock detector fails. Obermarck's distributed algorithm constructs a transaction-waits-for (TWF) graph at each site. Each site conducts deadlock detection simultaneously, passing

information to one other site. Deadlock detection activity at a site may become temporarily inactive until receipt of new information from another site. Obermarck states that in actual practice, synchronization (not necessarily precise) between sites would be roughly controlled by an agreed-upon interval between deadlock detection iterations, and by timestamps on transmitted messages. Nodes in the graph represent transactions, and edges represent a transaction-waits-for-transaction (TWFT) situation. A 'String' is a list of TWFT information which is sent from one site to one or more sites. A transaction may migrate from site to site, in which case an 'agent' represents the transaction at the new site(s). A communication link is also established between agents of a transaction. These communication links are represented by a node called 'External.' An agent which is expected to send a message is shown in the WF graph by EX—>T, while an agent waiting to receive is shown by T—>EX. Although Obermarck's algorithm includes the resolution of deadlocks, only the detection part will be considered in this paper. Transaction ID's are network unique names for transactions, and are lexically ordered. (For example, T1 < T2 < T3). The steps performed at each site are:

1. Build a TWF graph using transaction to transaction wait-for relationships.

2. Obtain and add to the existing TWF graph any 'strings' transmitted from other sites.

   a. For each transaction identified in a string, create a node in the TWF if none exists in this site.

   b. For each transaction in the string, starting with the first (which is always 'external'), create an edge to the node representing the next transaction in the string.

3. Create wait-for edges from 'external' to each node representing a transaction's agent which is expected to send on a communication link.

4. Create a WF edge from each node representing a transaction's

agent which is waiting to receive from a communication link, to 'external.'

5. Analyze the graph for cycles.

6. After resolving all cycles not involving 'external', if the transaction ID of the node for which 'external' waits is greater than the Transaction ID of the node waiting for 'external', then

   a. Transform the cycle into a string which starts with 'external', followed by each transaction ID in the cycle, ending with the transaction ID of the node waiting for 'external'.

   b. Send the string to each site for which the transaction terminating the string is waiting to receive.

In his proof of correctness, Obermarck shows how the algorithm can detect false deadlocks because a string received at a site may no longer be valid when it is used. He discusses two methods of handling false deadlocks; treat them as actual deadlocks(if they don't occur too often), or verify them by sending them around the network and have each site verify them.

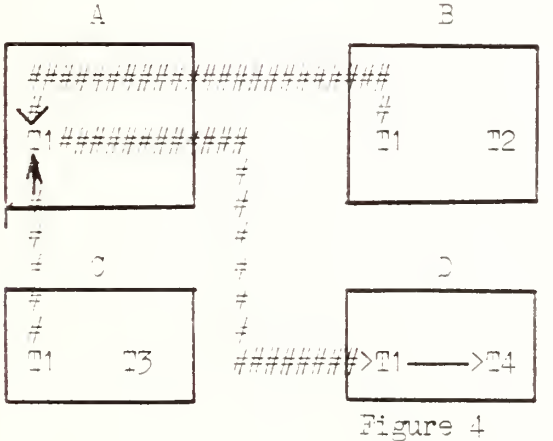


Figure 4

Figure 4 shows a global picture of the system, including the communication links established between agents, for the initial conditions of our example. The agents of T1 at sites B and C have

performed work (used R2 and R3), and are waiting for the next request from T1 at site A. T1 at site A is waiting for its agent at site D, which is in resource-wait for T4. Figure 5 shows the actions of this algorithm in an environment of no errors. As can be seen, it successfully detects the deadlock.

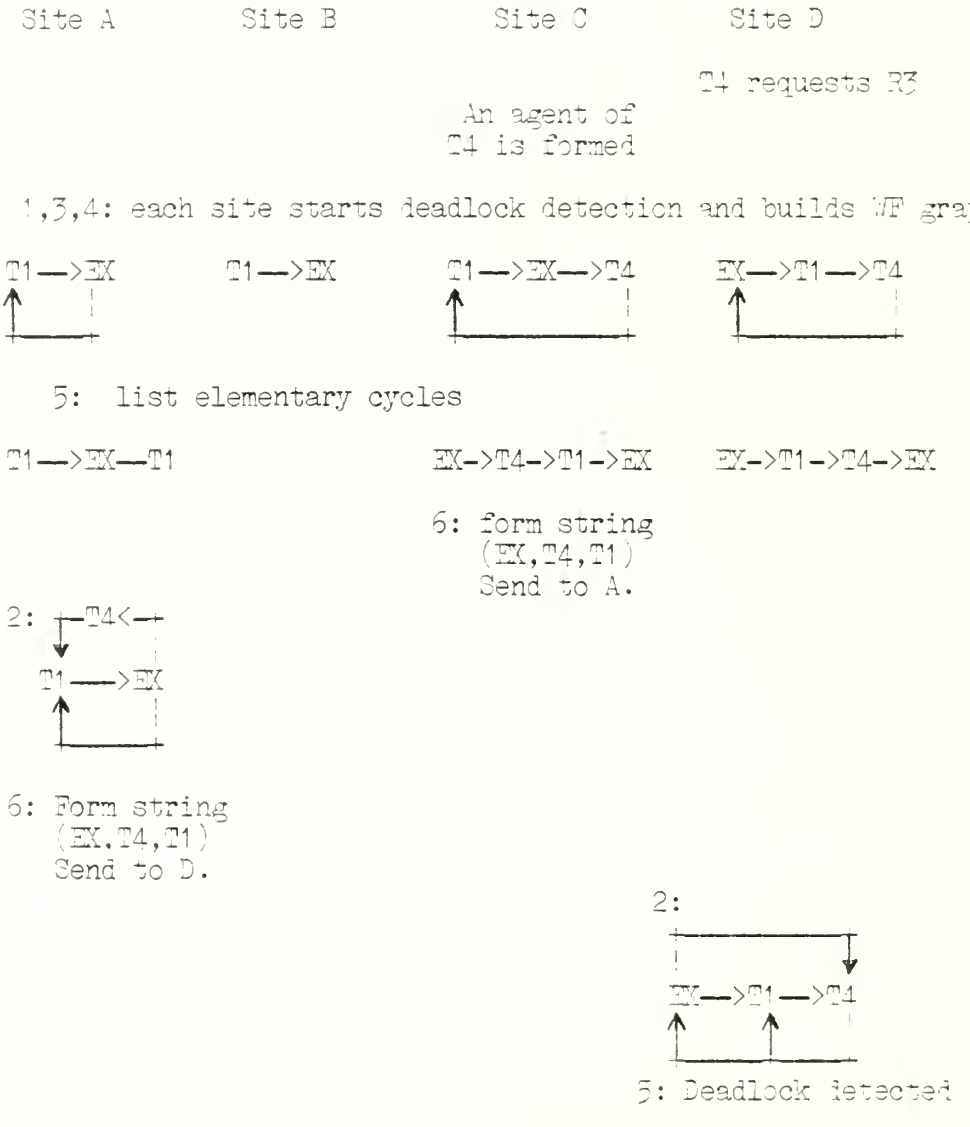


Figure 5

Obermarck assumes that messages sent are received. This is essential to the correctness of this DDA, because it is easy to see what happens if a message is lost. If the string (EX,T4,T1) from site C to A, or from A to D were lost, deadlock detection activity would

cease without detecting the deadlock. The use of agents to represent transactions which have migrated to other sites allow this DDA to have nodes of types a or b, if we substitute 'agents' for 'transactions' in our definitions at the beginning of this section. Site B would be an example of a type a site, while the other three sites would all be type b sites.

A failure in site B would have no effect on the behavior of the DDA. A failure at sites A, B or D would either have no effect, an undetermined effect, or cause deadlock detection activity to cease, depending on the time of the failure. For example, if site C failed before sending the string (EX,T4,T1) to site A, deadlock detection activity would cease. If site A (or D) failed before the string (EX,T4,T1) was sent to them, the transmitting site would recognize the failure, but its action in that eventuality is not included in the steps of the DDA. If site C failed after sending the string, the detection activity would continue, and the deadlock would be detected.

This DDA appears to be potentially more robust than the previous two. Each site contains and retains more information in its WF graph, and all sites start detection activity simultaneously, and potentially stay involved for the entire detection process. The use of the lexical ordering of nodes was for optimization of the number of messages transmitted. If this constraint were lifted, the strings would be sent to all sites involved from all sites in which a cycle existed. In our example, this would have allowed sites A and D to simultaneously detect deadlock. The DDA would be clearly more robust, but the overhead would be greater. In its existing form, this DDA's robustness is similar to the previous algorithms because it is essentially sequentially detecting the deadlock.

D. THE ALGORITHM OF TSAI AND BELFORD.

In [TSA82], Tsai and Belford present a distributed deadlock detection algorithm. They utilize a "Reduced Transaction-Resource" (RTR) graph, which contains only a subset of the transaction resource graph, but has all relevent TWF edges. Nodes in the RTR graph can be transactions or resources. The algorithm uses a concept the authors call a "reaching pair", which is the basic unit of information passed from site to site. If a path TiTj...Tn can be formed by following TWF edges, and if there is a request edge (Tn,Rm), then Ti "reaches" Rm, and (Ti,Rm) is a "reaching pair." Five types of messages are sent between sites: reaching messages, nonlocal request messages, allocation messages, release-request messages, and releasing messages. The non-local request messages include a list of all resources currently held by the requesting transaction. Five different types of edges are distinquished in the RTR graph: requesting edges, allocation edges, TWF edges, resource reaching edges and transaction reaching edges. A global timestamp is also used to establish an ordering of events. This timestamp is used on allocation, request and reaching messages, and on allocation and reaching edges in the RTR graph. The notation used in the algorithm is:

TS(M): timestamp of a message
TS(C): current system time
TS(A): timestamp of an allocation edge
TS(R): timestamp of a reaching edge
=/=: not equal to
Sorig: Site of origin

The steps of the algorithm (as executed at site Sk) are:

Step 1: {A transaction T enters the system requesting a nonlocal resource R} Add request edge (T,R) to RTR graph. Send request message (T,R',R,TS) to Sorig(R), where R' is the set of all resources allocated to T, and TS(M) = TS(C). R' has each TS(A) attached, and R' is empty if T holds no resources.

Step 1a: {A transaction T releases a nonlocal resource R} Erase edge (R,T) in the RTR graph. Send a release-request message(R,T) to Sorig(R).

Step 2: {A transaction T enters system requesting local resource R} Go to step 4.

Step 2a: {A transaction T releases a local resource R} Erase edge(R,T) in RTR graph. If there is any transaction T' waiting for R, then begin
Add allocation edge (R,T') to RTR graph with TS(A) = TS(C). Send allocation message (R,T',TS) with TS(M) = TS(C) to Sorig(T') if Sorig(T') =/= Sk. end.

Step 3: {A request message (T,R',R,TS) is received} Add allocation edges (Ri,T) for each Ri in R' to RTR graph. Go to step 4.

Step 3a: {A release-request message (R,T) is received} Erase allocation edge (R,T) in RTR graph. Send releasing message (R,T) to Sorig(T). If there is any transaction T' waiting for R, then begin
Add allocation edge (R,T') to RTR graph with TS(A) = TS(C). Send allocation message (R,T',TS) to Sorig(T') if Sorig(T') =/= Sk. end.

Step 4: If R is not held by any transaction, then begin
Add allocation edge (R,T) with TS(A)=TS(C) to RTR graph. If Sorig(T) =/= Sk, then send an allcation message (R,T,TS) with TS(M)=TS(C) to Sorig(T). end.
else begin
Add requesting edge (T,R) to RTR graph. Suppose R is held by transaction T'. Add edge (T,T') to RTR graph. If there is a cycle, deadlock has been detected, else go to step 5. end.

Step 5: {reaching message generation step} If there are two edges (T,R) and (T,T') added to the graph, and if T'T"...T" is any path obtained by following the TWF and transaction reaching edges, then set X =R" if T" has outgoing edge to R", else set X = R. For all transaction Ti in RTR graph reaching X via T, do begin
If Ti holds any resource R' with Sorig(Ti) =/= Sorig(R') and Sorig(R') =/= Sk, then send a reaching message (Ti,X,TS) to Sorig(R'). If Sorig(Ti) =/= Sk and Ti =/= T, then send a reaching message (Ti,X,TS) to Sorig(Ti). If Sorig(Ti) =/= Sk and Ti = T and X = R" then send a reaching message (Ti,X,TS) to Sorig(Ti). The TS in the reaching message is set to TS(C) if triggered by a local request, and set to TS(M) of the nonlocal request or reaching message otherwise.

Step 6: {An allocation message (R,T,TS) is received} If R is an entry in the graph, then begin
Erase allocation edge (R,T') and all reaching edges (T",R) with TS(R) < TS(M) and the corresponding TWF edge (T,T') and transaction reaching edges (T",T'), if

they exist, where T' =/= T. Change requesting edge (T,R) to allocation edge (R,T) with TS(A) = TS(M) if (T,R) exists, and for each resource reaching edge (T",R), add the transaction reaching edge (T",T). If Sorig(T) = Sk, wake up transaction T. end.

Step 6a: {A releasing message (R,T) is received} If Sorig(T) = Sk, wake up transaction T.

Step 7: {A reaching message (T,R,TS) is received} If there exists an allocation edge (R,T') in the graph with TS(M) < TS(A) and T' =/= T, then skip this step, else begin
Add resource reaching edge (T,R) to the RTR graph. If R is held by transaction T', then add the transaction reaching edge (T,T') to the graph. If there is a cycle in the graph, there is deadlock (go to step 8), otherwise go to step 5. end.

Step 8: {a deadlock has been detected} Take appropriate action.

Figure 6 shows the starting WF graphs and the actions of the DDA resulting from the request by transaction T4 for resource R3. An important item to note is that as soon the request is made, step 1 adds sufficient information to the WF graph to detect a deadlock, but does not check for deadlock, so the request is sent to site C and the algorithm continues. The obvious thing to do would be to add a check for a deadlock cycle in step one, but on closer analysis, this check may lead to detection of false deadlocks (if, for example, T1 had just released R3 but the message had not yet been received by site D.) Therefore the algorithm in its present form will be analyzed. The only message sent by this algorithm in this example is the request message (T4,{R4},R3,t6). If it was lost, the current algorithm would cease detection activity without detecting deadlock. In this instance, if the algorithm checked for deadlock in step 1, it would have been detected with no messages required.

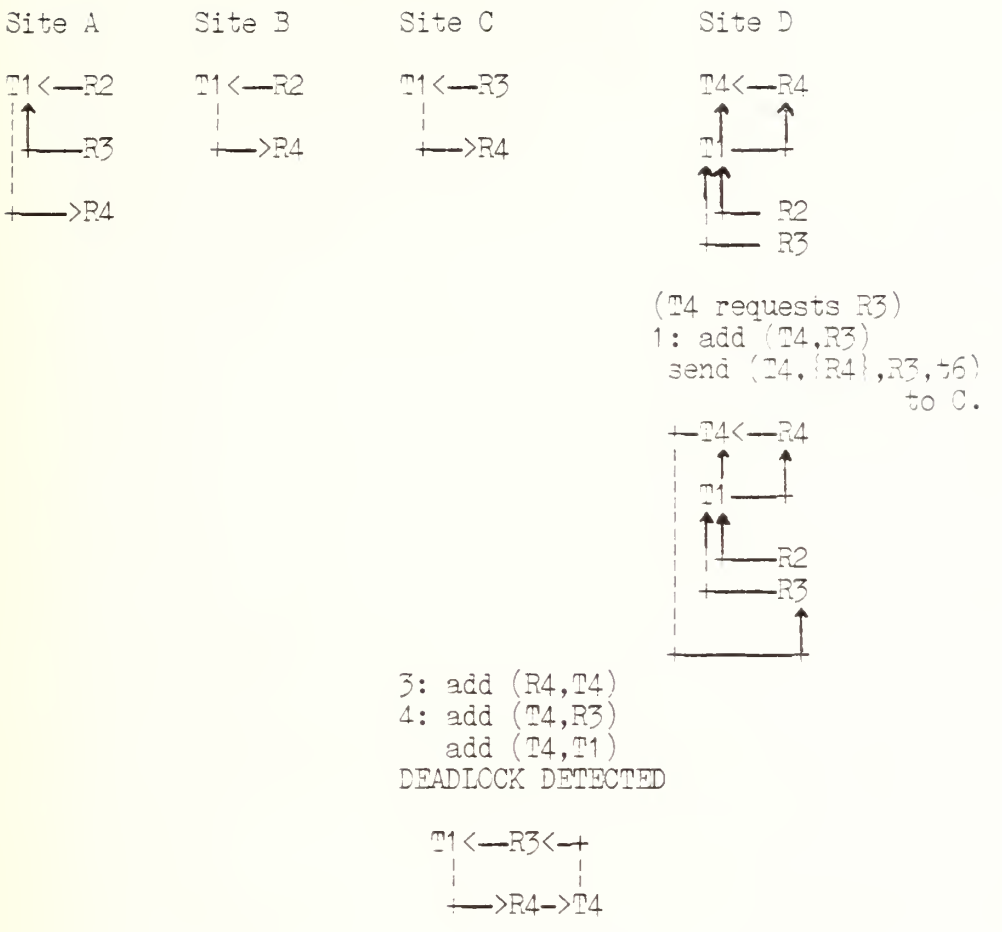


Figure 6

For this DDA, sites can be of type b, d or e. Sites A and D are type b and sites B and C are type d. This example has no type e sites, but step 5 of the algorithm could send reaching messages to sites not involved at all. Those sites would execute a step or two of the algorithm, but not be intimately involved in the actual deadlock detection. In this example, a failure of sites A or B (types b and d respectively) would have no effect on the detection of the deadlock. The effect of a failure of site C before the reaching message was sent to it cannot be determined because the DDA includes no instructions for that event. A failure of site C after receiving the reaching message would result in a cessation of detection activity. If the algorithm were modified to include a cycle check in step 4, a failure of

site C at any time would have no effect on deadlock detection. The timing of the failure would also determine the behavior of the DDA if site D failed. If site D failed before sending the request message, detection activity would cease, while if the message had been sent, deadlock would still be detected.

For our example, this DDA appears to be about the same level of robustness as the other algorithms, except that each site contains and retains more information than in other DDA's. This indicates that it should be more robust. The algorithm in the case of our example was able to detect the deadlock with only the resource request message. As deadlock cycles become more complex, it appears that this algorithm will also become more robust, even more so than Obermarck's, because this DDA retains more information, and it will send reaching messages to any site potentially involved in the deadlock. Detection activity will occur simultaneously in those sites receiving reaching messages. The impact of the inclusion of a cycle detection in step 1 may have adverse effects on the correctness, but it might greatly enhance the robustness of the DDA.

IV. CONCLUSIONS

The algorithms discussed in the previous section can be loosely ranked by their robustness. Goldman's algorithm is the least robust, because it is always executed sequentially (unless the requests occur simultaneously, as discussed previously). Thus it is always dependent on a single node. Obermarck's algorithm starts deadlock detection simultaneously at all sites, and subsequently passes information in a lexical manner because of the message optimization. For our example, this resulted in a sequential detection, although for larger deadlock

cycles, it should have some parallel detection activity occuring. The Menasce-Muntz algorithm starts detection at the site where the deadlock occured, and deadlock detection is subsequently conducted at sites which are potentially involved. The Tsai-Belford algorithm is invoked each time a rescurce is requested. Deadlock detection can appear concurrently at all sites potentially involved in the cycle. It appears more robust than the Menasce-Muntz algorithm because more information is held at each site.

Our analysis supports the rather obvious conclusion that robustness is inversely related to it's cost. The Tsai-Belford algorithm appears more robust than Obermarck's algorithm, for example, but it maintains larger WF graphs at each site, and is invoked each time a resource is requested, in order that the WF graphs contain sufficient information.

For the example we used to analyze the four algorithms in section 3, the behavior of each of those algorithms in the presence of errors is almort identical. Because our deadlock cycle only involved 2 transactions, those algorithms which are potentially more robust in the presence of larger cycles did not have time to develop their robustness. In other words, for a short deadlock cycle, all the algorithms converged within approximately the same length of time (two or three iterations.) Short cycles of length 2 or 3 are more probable in existing applications, so all the above algorithms are approximately equally robust in current applications. In future applications (information utility programs, for example), however, we expect a much higher probability of more complex deadlock cycles, which will require a more robust DDA. Conversely, however, as the number of transactions (and sites) increases, it will be important to use a minimum cost DDA.

Work is currently in progress on a new robust distributed deadlock detection algorithm.

References

[BEE81] C. Beeri and R. Obermarck. A Resource Class Independent Deadlock Detection Algorithm. Presented at Seventh International Conference on Very Large Data Bases, September 1981.

[GLI80] V. Gligor and S. Shattuck. "On Deadlock Detection in Distributed Systems." IEEE Trans on Software Eng, Vol SE-6, September, 1980. pp 435-440.

[GOL77] B. Goldman. "Deadlock Detection in Computer Networks." Technical report TR-MIT/LCS/TR-185, Lab of CS, MIT, September, 1977.

[GRA78] J. Gray. "Notes on Data Base Operating Systems." Research Report, IBM Research Division RJ2188(30001), February, 1978.

[GRA81] J. Gray. "The Transaction Concept: Virtues and Limitations," Tandem TR81.3, June 1981.

[ISL78] S. Isloor and T. Marsland. "An Effective 'On-line' Deadlock Detection Technique for Distributed Data Base Management Systems," in Proc. COMSAC 1978, pp 283-288.

[MEN79] D. Menasce and R. Muntz. "Locking and Deadlock Detection in Distributed Data Bases," IEEE Trans on Software Eng, Vol SE-5, No. 3, May, 1979, pp. 195-202.

[OBE80] R. Obermarck. "Global Deadlock Detection Algorithm." Research Report, IBM Research Division RJ2845(36131), June 1980.

[TSA82] Tsai and G. Belford. "Detecting Deadlock in a Distributed System." To appear in Proc INFOCOM, April 1, 1982.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	30
Dushan Badal, Code 52Zd Department of Computer Science Naval Postgraduate School Monterey, CA 93940	5
Robert B. Grafton Office of Naval Research Code 437 800 N. Quincy Street Arlington, VA 22217	1
David W. Mizell Office of Naval Research 1030 East Green Street Pasadena, CA 91106	2
Vinton G. Cerf DARPA/IPTO 1400 Wilson Blvd Arlington, VA 22209	1
CDR R. Ohlander DARPA 1400 Wilson Blvd Arlington, VA 22209	1
Col. D. Adams DARPA 1400 Wilson Blvd Arlington, VA 22209	1

CAPT. W. Price 1
AFOSR/NM
Bolling AFB., D.C. 20332

Col. R. Schell 1
National Security Agency
C1
Fort George Meade, MD 20755

Raymond A. Liuzzi 1
RADC/COTD
Griffiss AFB., N.Y. 13441

Thomas Lawrence 1
RADC/COTD
Griffiss AFB., N.Y. 13441

LCDR Michael T. Gehl 5
SMC 2405
Naval Postgraduate School
Monterey, CA 93940

Commander, Naval Sea Systems Command 2
Sea 00Z
Naval Sea Systems Command Headquarters
Washington, D.C. 20362